Docket No.: BBNT-P01-287
(PATENT)

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent Application of:
Weinstein et al.

Application No.: 09/546052 Art Unit: 2665

Filed: April 10, 2000 Examiner: T. D. Nguyen

For: RADIO NETWORK ROUTING APPARATUS

DECLARATION UNDER 37 C.F.R. § 1.131

I, Joseph J. Weinstein, do hereby declare as follows:

1. I am one of the inventors of the subject matter claimed and described in the above referenced patent application.

2. I discussed the legal meaning of the terms "conceived" and "reduced to practice" with my attorney. Based on my understanding of those terms derived from that discussion, I, with my coinventors, conceived of and reduced to practice, in the United States, the subject matter of the pending claims prior to November 12, 1999.

3. The computer source code, attached as Exhibit A, was entered into the code archive system of Assignee, BBNT Solutions LLC, prior to November 12, 1999. Dates, comments, and copyright information have been redacted from the source code. The source code illustrates the relevant portions of a computer program as described in claims 46–54 that carries out the methods described in claims 21–28. Such code was operated on at least one router prior to November 12, 1999, as described in claims 37–45 to yield a system described in claims 29–36. The code illustrates one embodiment of the invention and is not intended to narrow the scope of the claims.

4. In particular, the attached code includes the following functions and procedures:

a. rospf_open_fwd_connection, rospf_close_fwd_connnection, rospf_get_intranet_topology, rospf_get_next_entry, together enable a router to obtain lower layer topology information from a lower layer protocol, in this case, the NTDR (near term digital radio) protocol.

b. ospf_states.c, a modified version of the ospf_states.c file from the GATED consortium, calls the rospf_get_intranet_topology and rospf_get_next_entry routines to use in forming higher layer protocol (in this case the OSPF protocol) adjacencies.

c. tq_ntdr_fwd_handler provides for a timer for initiating a periodic higher layer protocol adjacency updating process which includes obtaining updated lower layer protocol topology information.

5. I assert that all statements made of my own knowledge are true, and that all statements made on information and belief are believed to be true. I also understand that willful false statements and the like are punishable by fine or imprisonment, or both (18 U.S.C. § 1001) and may jeopardize the validity of the application or any patent issuing thereon.

Dated: Feb. 17, 2005

Joseph J. Weinstein

Ex. A1

REDACTED

```
#ifndef ROSPF_FWD_INTERFACE_H
#define ROSPF_FWD_INTERFACE_H

#include "include.h"

#ifdef ROSPF
#include <syst/nfwd.h>

#define ROSPF_MAX_NET_SIZE 1023
#define ROSPF_AVG_NET_SIZE 200

boolean_t   rospf_open_fwd_connection(struct INTF * inIntf);
void        rospf_close_fwd_connection(struct INTF * inIntf);

boolean_t   rospf_get_intranet_topology(struct INTF * inIntf);
void        rospf_reset_intranet_topology(void);
boolean_t   rospf_get_next_entry(struct INTF * inIntf,nfwd_flag_desc_t ** outEntry);

struct NBR *        rospf_find_nbr_by_addr(struct INTF * inIntf, u_int32 inNbrAddr);
struct NBR *        rospf_find_nbr_by_id(struct INTF * inIntf, u_int32 inNbrId);
nfwd_flag_desc_t  * rospf_find_entry_by_addr(struct INTF * inIntf,u_int32 inNbrAddr);

u_int32 ROSPF_CONVERT_MAC_TO_IP(struct INTF * inIntf,u_int32 inMac);
u_int32 ROSPF_CONVERT_IP_TO_MAC(struct INTF * inIntf,u_int32 inIp);

void rospf_save_cluster_role(struct NBR * inNbr,nfwd_flag_desc_t * inRoute);
boolean_t rospf_is_role_different(struct NBR * inNbr,nfwd_flag_desc_t * inNode);



#endif
#endif
```

Ex A2

REDACTED

```
#include "include.h"
#include "ospf.h"
#include "inet.h"


#ifdef ROSPF

#define ROSPF_DEBUG 1

#include "rospf_fwd_interface.h"
#include <stropts.h>
#include <sys/types.h>
#include <syst/nfwd.h>
#include <syst/ntdr_stropts.h>
#include <fcntl.h>

static struct strioctl sIoctl;                  /* structure used for polling NTDR forwarding table
*/
static nfwd_get_flags_t *  sBuffer     = NULL;        /* buffer to hold polled NTDR Forwarding
table */
static int             sCurIndex    = -1;          /* current element in sBuffer being used */
static int             sNumElements = 0;           /* current number of elements(neighbors) in
sBuffer */
static int             sBufferSize  = 0;           /* Capacity (in neighbors) of sBuffer */



int compute_size(int inNumElements)
{
 return (sizeof(nfwd_get_flags_t) + ( (inNumElements-1) * sizeof(nfwd_flag_desc_t) ) );
}



boolean_t rospf_open_fwd_connection(struct INTF * inIntf)
{
 trace_tf(ospf.trace_options,
          TR_ROSPF,
          0,
          ("ROSPF: opening connection to ntdr forwarding module: %s",
           inIntf->rospf_fwd_dev));

 if ( (inIntf->rospf_fwd_fd = open(inIntf->rospf_fwd_dev,O_RDWR)) < 0 )
   {
    trace_log_tf(ospf.trace_options,
```

```
                    0,
                    LOG_WARNING,
                    ("ROSPF: Open failed on device %s",
                     inIntf->rospf_fwd_dev));
     assert(FALSE);
    }

  if (sBuffer == NULL)
    {
     trace_tf(ospf.trace_options,
              TR_ROSPF,
              0,
              ("ROSPF: Allocating Buffer for ntdr forwarding Table"));
     sBufferSize = ROSPF_AVG_NET_SIZE;
     sBuffer     = (nfwd_get_flags_t *) task_mem_malloc(NULL,compute_size(sBufferSize));

     assert(sBuffer!=NULL);
    }
  return(B_TRUE);
}



void rospf_close_fwd_connection(struct INTF * inIntf)
{
 close(inIntf->rospf_fwd_fd);
}



boolean_t rospf_get_intranet_topology(struct INTF * inIntf)
{
 boolean_t isDone;
 int       theNumEntries;


 isDone = B_FALSE;

 while (isDone == B_FALSE)
   {

    sIoctl.ic_cmd   = nfwd_GET_ALL_FLAGS;
    sIoctl.ic_timout = 10;
    sIoctl.ic_len   = compute_size(sBufferSize);
```

```
    sIoctl.ic_dp    = (char *) sBuffer;


    if ( (ioctl(inIntf->rospf_fwd_fd,I_STR,&sIoctl) != 0) &&
         ( sBuffer->nentries <= sBufferSize ) )
        {
         trace_tf(ospf.trace_options,
                  TR_ROSPF,
                  0,
                  ("ROSPF: ioctl failed for getting forwarding table "));
         assert(FALSE);
        }

    if ( sBuffer->nentries > sBufferSize )
        {
         trace_tf(ospf.trace_options,
                  TR_ROSPF,
                  0,
                  ("ROSPF: ioctl failed, buffer too small, nentries: %d sBufferSize:%d, re-
allocating",
                   sBuffer->nentries,sBufferSize));
         theNumEntries = sBuffer->nentries;
         task_mem_free((task *) NULL,sBuffer); sBuffer = NULL;
         sBufferSize = theNumEntries + ( (ROSPF_MAX_NET_SIZE - theNumEntries) / 2 );

         sBuffer = task_mem_malloc((task *)NULL,compute_size(sBufferSize));
         assert(sBuffer != NULL);
        }
    else
        {
         trace_tf(ospf.trace_options,
                  TR_ROSPF,
                  0,
                  ("ROSPF: forwarding table has %d entries",
                   sBuffer->nentries));

         isDone      = B_TRUE;
         sCurIndex   = -1;
         sNumElements = sBuffer->nentries;
        }
  }

 return(B_TRUE);
}
```

```
void rospf_reset_intranet_topology(void)
{
 sCurIndex    = -1;
}


boolean_t rospf_get_next_entry(struct INTF * inIntf,nfwd_flag_desc_t ** outEntry)
{
 boolean_t isDone;

 isDone   = B_FALSE;

 while (isDone == B_FALSE)
  {
   sCurIndex++;

   if (sCurIndex >= sNumElements)
       {
        return(B_FALSE);
       }

   if((sBuffer->entries[sCurIndex].flags & nfwd_rlflags_NBR_DIRECT) &&
       (sBuffer->entries[sCurIndex].flags & nfwd_rlflags_NBR_MEMBER) &&
       (sBuffer->entries[sCurIndex].flags & nfwd_rlflags_CURNODE_MEMBER ) )
       {
        ;
       }
   else
       {
        isDone   = B_TRUE;
        *outEntry = &sBuffer->entries[sCurIndex];
        return(B_TRUE);
       }
  }
}


struct NBR * rospf_find_nbr_by_id(struct INTF * inIntf, u_int32 inNbrId)
{
 struct NBR        *   theCurNbr;
 struct LSDB_HEAD  *   theHead;
 struct LSDB       *   theCurAdv;
 struct RTR_LA_PIECES * theCurAdvIntf;
 int               theCurIntfIndex;
```

```
int             theIntfIndexMax;
boolean_t       foundAdv = B_FALSE;
struct RTR_LA_HDR *   theAdvHeader;


theCurNbr = FirstNbr(inIntf);
while (theCurNbr != NULL)
 {
  if ( NBR_ID(theCurNbr) == inNbrId)
      {
       return(theCurNbr);
      }
  theCurNbr = theCurNbr->next;
 }


LSDB_HEAD_LIST(inIntf->area->htbl[LS_RTR], theHead, 0, HTBLSIZE)
 {
  LSDB_LIST(theHead, theCurAdv)
      {
       theAdvHeader = (struct RTR_LA_HDR *) theCurAdv->lsdb_adv.rtr;

       if ( (inNbrId == ADV_RTR(theCurAdv)) && (theAdvHeader->ls_hdr.ls_type ==
LS_RTR) )
        {
         theCurIntfIndex = 0;
         theCurAdvIntf = & theAdvHeader->link;
         while (theCurIntfIndex < ntohs(theAdvHeader->lnk_cnt) )
             {
              if ((theCurAdvIntf->lnk_data & INTF_MASK(inIntf)) ==
                INTF_NET(inIntf))
               {
                foundAdv = B_TRUE;

#ifdef ROSPF_DEBUG
                trace_tf(ospf.trace_options,
                         TR_ROSPF,
                         0,
                         ("ROSPF: Adv found in database. theCurAdvIntf.  lnk_id %A
lnk_data %A rtr_id %A",
                            sockbuild_in(0,theCurAdvIntf->lnk_id),
                            sockbuild_in(0,theCurAdvIntf->lnk_data),
                            sockbuild_in(0, ADV_RTR(theCurAdv))));
#endif
                break;
               }
```

```
                theCurIntfIndex++;
                theCurAdvIntf++;
              }
          }

         if ( foundAdv == B_TRUE)
          {
           break;
          }
        } LSDB_LIST_END(theHead, theCurAdv) ;

    if ( foundAdv == B_TRUE)
        {
         break;
        }
   } LSDB_HEAD_LIST_END(inIntf->area->htbl[LS_RTR], theHead, 0, HTBLSIZE) ;


 if (!foundAdv) return(NULL);

#ifdef ROSPF_DEBUG
 trace_tf(ospf.trace_options,
          TR_ROSPF,
          0,
          ("ROSPF: Adv found in database. theCurAdvIntf. lnk_id %A lnk_data %A rtr_id %A",
           sockbuild_in(0,theCurAdvIntf->lnk_id),
           sockbuild_in(0, ADV_RTR(theCurAdv))));
#endif

 theCurNbr = FirstNbr(inIntf);
 while (theCurNbr != NULL)
  {
   if (NBR_ADDR(theCurNbr) == theCurAdvIntf->lnk_data)
        {
         theCurNbr->nbr_id =
          sockdup(sockbuild_in(0, ADV_RTR(theCurAdv)));
         ospf_nbr_remove(inIntf, theCurNbr);
         ospf_nbr_add(inIntf, theCurNbr);
#ifdef ROSPF_DEBUG
         trace_tf(ospf.trace_options,
                  TR_ROSPF,
                  0,
                  ("ROSPF: NBR found nbr_id %A  nbr_add %A ",
                   theCurNbr->nbr_id,theCurNbr->nbr_addr));
#endif
         return(theCurNbr);
```

```
        }
     theCurNbr = theCurNbr->next;
    }

  return(NULL);

}


struct NBR * rospf_find_nbr_by_addr(struct INTF * inIntf, u_int32 inNbrAddr)
{
  struct NBR      * theCurNbr;
  struct LSDB_HEAD * theHead;
  struct LSDB     * theCurAdv;
  struct RTR_LA_PIECES * theCurAdvIntf;
  int             theCurIntfIndex;
  int             theIntfIndexMax;
  boolean_t       foundAdv = B_FALSE;
  struct RTR_LA_HDR * theAdvHeader;


  theCurNbr = FirstNbr(inIntf);
  while (theCurNbr != NULL)
    {
     if ( NBR_ADDR(theCurNbr) == inNbrAddr)
         {
          return(theCurNbr);
         }
     theCurNbr = theCurNbr->next;
    }


  LSDB_HEAD_LIST(inIntf->area->htbl[LS_RTR], theHead, 0, HTBLSIZE)
    {
     LSDB_LIST(theHead, theCurAdv)
         {

          theAdvHeader = (struct RTR_LA_HDR *) theCurAdv->lsdb_adv.rtr;

          if (theAdvHeader->ls_hdr.ls_type == LS_RTR)
            {
             theCurIntfIndex = 0;
             theCurAdvIntf  = &theAdvHeader->link;

             while (theCurIntfIndex < ntohs(theAdvHeader->lnk_cnt) )
                 {
```

```
                if (theCurAdvIntf->lnk_data == inNbrAddr)
                  {
#ifdef ROSPF_DEBUG
                    trace_tf(ospf.trace_options,
                             TR_ROSPF,
                             0,
                             ("ROSPF: find nbr by addr - Adv found in database. theCurAdvIntf
lnk_id %A lnk_data %A rtr_id %A",
                               sockbuild_in(0,theCurAdvIntf->lnk_id),
                               sockbuild_in(0,theCurAdvIntf->lnk_data),
                               sockbuild_in(0, ADV_RTR(theCurAdv))));
#endif
                    foundAdv = B_TRUE;
                    break;
                  }
                theCurIntfIndex++;
                theCurAdvIntf++;
              }
          }

         if ( foundAdv == B_TRUE)
           {
            break;
           }
        } LSDB_LIST_END(theHead, theCurAdv) ;

    if ( foundAdv == B_TRUE)
        {
         break;
        }
   } LSDB_HEAD_LIST_END(inIntf->area->htbl[LS_RTR], theHead, 0, HTBLSIZE) ;


 if (!foundAdv) return(NULL);

#ifdef ROSPF_DEBUG
 trace_tf(ospf.trace_options,
          TR_ROSPF,
          0,
          ("ROSPF: find nbr by addr - Adv found in database. theCurAdvIntf lnk_id %A
lnk_data %A rtr_id %A",
           sockbuild_in(0,theCurAdvIntf->lnk_id),
           sockbuild_in(0,theCurAdvIntf->lnk_data),
           sockbuild_in(0, ADV_RTR(theCurAdv))));
#endif
```

```
 theCurNbr = FirstNbr(inIntf);
 while (theCurNbr != NULL)
  {
   if (NBR_ID(theCurNbr) == ADV_RTR(theCurAdv))
       {
        theCurNbr->nbr_addr =
          sockdup(sockbuild_in(0, theCurAdvIntf->lnk_data));
        return(theCurNbr);
       }
   theCurNbr = theCurNbr->next;
  }

 return(NULL);

}

nfwd_flag_desc_t  * rospf_find_entry_by_addr(struct INTF * inIntf,u_int32 inNbrAddr)
{
 u_int32   theMac;
 boolean_t isDone;
 int       theIndex;

 theMac   = ROSPF_CONVERT_IP_TO_MAC(inIntf,inNbrAddr);
 isDone   = B_FALSE;
 theIndex = 0;

 while (isDone == B_FALSE)
  {
   if (theIndex >= sNumElements)
       {
        return(NULL);
       }

   if(sBuffer->entries[theIndex].destination == theMac)
       {
        return(&sBuffer->entries[theIndex]);
       }
   theIndex++;
  }
}


u_int32 ROSPF_CONVERT_MAC_TO_IP(struct INTF * inIntf, u_int32 inMac)
{
 u_int32 theHostNet;
```

```
 theHostNet = ntohl(INTF_NET(inIntf));

 return( htonl(theHostNet | (inMac)) );
}

u_int32 ROSPF_CONVERT_IP_TO_MAC(struct INTF * inIntf,u_int32 inIp)
{
 u_int32 theHostAddr;
 u_int32 theHostNet;

 theHostAddr   = ntohl(inIp);
 theHostNet    = ntohl(INTF_NET(inIntf));

 return ( theHostAddr & ~theHostNet);
}


void rospf_save_cluster_role(struct NBR * inNbr,nfwd_flag_desc_t * inRoute)
{
 if ( inRoute->flags & nfwd_rlflags_NBR_HEAD )
   {
    inNbr->nbr_flags |= NBR_RADIO_NBR_HEAD;
   }

 if ( inRoute->flags & nfwd_rlflags_NBR_MEMBER )
   {
    inNbr->nbr_flags |= NBR_RADIO_NBR_MEMBER;
   }

 if ( inRoute->flags & nfwd_rlflags_CURNODE_HEAD )
   {
    inNbr->nbr_flags |= NBR_RADIO_CURNODE_HEAD;
   }

 if ( inRoute->flags & nfwd_rlflags_CURNODE_MEMBER)
   {
    inNbr->nbr_flags |= NBR_RADIO_CURNODE_MEMBER;
   }
}


boolean_t rospf_is_role_different(struct NBR * inNbr,nfwd_flag_desc_t * inRoute )
{
 if ( ( inRoute->flags & nfwd_rlflags_NBR_HEAD )
    && !(inNbr->nbr_flags & NBR_RADIO_NBR_HEAD))
   {
```

```
    return(B_TRUE);
   }

  if ((inRoute->flags & nfwd_rlflags_NBR_MEMBER)
    && !(inNbr->nbr_flags & nfwd_rlflags_NBR_MEMBER))
   {
    return(B_TRUE);
   }

  if ((inRoute->flags & nfwd_rlflags_CURNODE_HEAD)
    && !(inNbr->nbr_flags & NBR_RADIO_CURNODE_HEAD))
   {
    return(B_TRUE);
   }

  if ((inRoute->flags & nfwd_rlflags_CURNODE_MEMBER)
    && !(inNbr->nbr_flags & NBR_RADIO_CURNODE_MEMBER))
   {
    return(B_TRUE);
   }
  return(B_FALSE);
}



#endif
```

ospf_states.c

```
#defineINCLUDE_TIME
#include "include.h"
#include "inet.h"
#include "ospf.h"

#ifdef ROSPF
#include <syst/nfwd.h>
#endif

/*****************************************************************************
*
 *
 *                    STATE TRANSITION SUPPORT ROUTINES
 *

*****************************************************************************
/



/*****************************************************************************
*
 *
 *                    NEIGHBOR STATE TRANSITIONS
 *

*****************************************************************************
/

static const char *ospf_nbr_events[] = {
   "Hello Received",
   "Start",
   "Two Way Received",
   "Adjacency OK",
   "Negotiation Done",
   "Exchange Done",
   "Sequence # Mismatch",
   "Bad LS Request",
   "Loading Done",
   "One way",
   "Reset Adjacency",
   "Kill Neighbor",
```

```
   "Inactivity Timer",
#ifdef ROSPF
   "Lower Level Down",
   "Lower Level Up"
#else
   "Lower Level Down"
#endif
};

const char *ospf_nbr_states[] = {
   "Down",
   "Attempt",
   "Init",
   "Two Way",
   "Exch Start",
   "Exchange",
   "Loading",
   "Full",
   "SCVirtual"
};

#definemsg_event_nbr(nbr, event, old) \
        if (TRACE_TF(ospf.trace_options, TR_STATE)) { \
           trace_only_tf(ospf.trace_options, \
                         TRC_NL_AFTER,\
                         ("OSPF TRANSITION       Neighbor %A  EVENT %s  %10s -> %-
10s", \
                          (nbr)->nbr_addr, \
                          ospf_nbr_events[event], \
                          ospf_nbr_states[old], \
                          ospf_nbr_states[(nbr)->state])); \
        }


static void
NErr __PF2(intf, struct INTF *,
           nbr, struct NBR *)
{
}



static void
NHello __PF2(intf, struct INTF *,
             nbr, struct NBR *)
{
```

```
    u_int oldstate = nbr->state;
    struct AREA *area = intf->area;

    u_int32 nh_addr = (NBR_ADDR(nbr)) ? NBR_ADDR(nbr) : NBR_ID(nbr);


    reset_inact_tmr(nbr);
    nbr->state = NINIT;
    intf->nbrIcnt++;
    ospf.nbrIcnt++;
    area->nbrIcnt++;
    OSPF_NH_ALLOC(nbr->nbr_nh = ospf_nh_add(nbr->intf->ifap,
                                            nh_addr,
                                            NH_NBR));

    msg_event_nbr(nbr, HELLO_RX, oldstate);
    nbr->events++;
}

static void
NStart __PF2(intf, struct INTF *,
             nbr, struct NBR *)
{
    u_int oldstate = nbr->state;

    nbr->state = NATTEMPT;
    if (intf->pri)
        send_hello(intf, nbr, FALSE);
    reset_inact_tmr(nbr);

    msg_event_nbr(nbr, START, oldstate);
    nbr->events++;
}


static void
N2Way __PF2(intf, struct INTF *,
            nbr, struct NBR *)
{

    u_int oldstate = nbr->state;

    nbr->state = N2WAY;

    if ((intf->type <= NONBROADCAST) && (intf->state > IPOINT_TO_POINT)) {
        (*(if_trans[NBR_CHANGE][intf->state])) (intf);
```

```
    }

    msg_event_nbr(nbr, TWOWAY, oldstate);
    nbr->events++;

    if ((intf->type > NONBROADCAST && intf->type <= VIRTUAL_LINK) ||
        ((nbr->state == N2WAY && intf->type < POINT_TO_POINT) &&
         (intf->dr == nbr || intf->bdr == nbr ||
          intf->dr == &intf->nbr || intf->bdr == &intf->nbr)))
        (*(nbr_trans[ADJ_OK][nbr->state])) (intf, nbr);

}


static void
NAdjOk __PF2(intf, struct INTF *,
             nbr, struct NBR *)
{
    u_int oldstate = nbr->state;

    nbr->state = NEXSTART;

    nbr->seq = time_sec;
    nbr->I_M_MS = (bit_I | bit_M | bit_MS);
    send_dbsum(intf, nbr, 0);

    msg_event_nbr(nbr, ADJ_OK, oldstate);
    nbr->events++;
}


static void
NNegDone __PF2(intf, struct INTF *,
               nbr, struct NBR *)
{
    u_int oldstate = nbr->state;
    struct AREA *area = intf->area;


    intf->nbrEcnt++;
    area->nbrEcnt++;
    ospf.nbrEcnt++;

    if (build_dbsum(intf, nbr)) {
        intf->nbrEcnt--;
        area->nbrEcnt--;
```

```
        ospf.nbrEcnt--;
        return;
    }

    nbr->state = NEXCHANGE;
    msg_event_nbr(nbr, NEGO_DONE, oldstate);
    nbr->events++;
}


static void
NExchDone __PF2(intf, struct INTF *,
                nbr, struct NBR *)

{
    u_int oldstate = nbr->state;


    if (nbr->mode == MASTER) {
        if (nbr->dbsum != LSDB_SUM_NULL)
            freeDbSum(nbr);
    } else {

        nbr->mode = SLAVE_HOLD;
        set_hold_tmr(nbr);
    }

    nbr->state = NLOADING;
    msg_event_nbr(nbr, EXCH_DONE, oldstate);
    nbr->events++;

    if (NO_REQ(nbr))
        (*(nbr_trans[LOAD_DONE][nbr->state])) (intf, nbr);
}


static void
NBadReq __PF2(intf, struct INTF *,
              nbr, struct NBR *)
{
    u_int oldstate = nbr->state;
    struct AREA *area = intf->area;


    freeDbSum(nbr);
    REM_NBR_RETRANS(nbr);
```

```
    freeLsReq(nbr);
    if (intf->nbrIcnt == 1) {
        freeAckList(intf);
    }

    if (nbr->state == NFULL) {
        intf->nbrFcnt--;
        area->nbrFcnt--;
        ospf.nbrFcnt--;
        if ((intf->type <= NONBROADCAST) && (intf->state >= IDr))
          BIT_SET(intf->flags, OSPF_INTFF_NBR_CHANGE);
        else if (intf->type == POINT_TO_POINT || intf->type == VIRTUAL_LINK)
          area->build_rtr = TRUE;

    }
    if (nbr->state >= NEXCHANGE) {
        intf->nbrEcnt--;
        area->nbrEcnt--;
        ospf.nbrEcnt--;
    }

    nbr->state = NEXSTART;
#ifdef  notdef
    intf->nbrEcnt--;
    area->nbrEcnt--;
    ospf.nbrEcnt--;
#endif

    nbr->seq = time_sec;
    nbr->I_M_MS = (bit_I | bit_M | bit_MS);
    send_dbsum(intf, nbr, 0);

    msg_event_nbr(nbr, BAD_LS_REQ, oldstate);
    nbr->events++;
}


static void
NBadSq __PF2(intf, struct INTF *,
             nbr, struct NBR *)
{
    u_int oldstate = nbr->state;
    struct AREA *area = intf->area;


    freeDbSum(nbr);
```

```
    REM_NBR_RETRANS(nbr);
    freeLsReq(nbr);

    if (intf->nbrIcnt == 1) {
        freeAckList(intf);
    }

    if (nbr->state == NFULL) {
        intf->nbrFcnt--;
        area->nbrFcnt--;
        ospf.nbrFcnt--;

        if ((intf->type <= NONBROADCAST) && (intf->state >= IDr))
          BIT_SET(intf->flags, OSPF_INTFF_NBR_CHANGE);
        else if (intf->type == POINT_TO_POINT || intf->type == VIRTUAL_LINK)
          area->build_rtr = TRUE;
    }
    if (nbr->state >= NEXCHANGE) {
        intf->nbrEcnt--;
        area->nbrEcnt--;
        ospf.nbrEcnt--;
    }

#ifdef ROSPF
    if (BIT_TEST(nbr->nbr_flags, NBR_RADIO_ACTIVE)) {
     assert(intf->state == IRADIO_MULTIPOINT);
     assert(intf->rospf_nbr_active_count > 0);

     intf->rospf_nbr_active_count--;
     BIT_RESET(nbr->nbr_flags, NBR_RADIO_ACTIVE);
     BIT_SET(intf->flags, OSPF_INTFF_NEW_NBR_ID_LEARNED);
     trace_log_tf(ospf.trace_options,
                  0,
                  LOG_INFO,
                  ("ROSPF: Nbr %A (id %A) now inactive",
                   nbr->nbr_addr,
                   nbr->nbr_id));

     if (!(BIT_TEST(intf->dr->nbr_flags, NBR_RADIO_ACTIVE)) &&
          NBR_ADDR(intf->dr) != INTF_ADDR(intf)) {
        struct NBR *theNbr;

        if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
         trace_tf(ospf.trace_options,
                  TR_ROSPF,
                  0,
```

```
                ("ROSPF: searching for new dr"));
        }

        theNbr  = FirstNbr(intf);
        while (theNbr != NULL)
         {
          if ( BIT_TEST(theNbr->nbr_flags,NBR_RADIO_ACTIVE) ||
               NBR_ADDR(theNbr) == INTF_ADDR(intf)) {

           assert(NBR_ADDR(theNbr) != (u_int32) 0);
           theNbr->dr = NBR_ADDR(theNbr);

           trace_log_tf(ospf.trace_options,
                        0,
                        LOG_INFO,
                        ("ROSPF: DR change from nbr addr %A (ID %A) to addr %A (ID
%A)",
                         intf->dr->nbr_addr,
                         intf->dr->nbr_id,
                         theNbr->nbr_addr,
                         theNbr->nbr_id));
           intf->dr = theNbr;
           rospf_spoof_net_lsa(intf);
           break;
          }
          theNbr = theNbr->next;
         }
     }
     set_rospf_buildnet_tmr(intf, ROSPF_BUILDNET_INTERVAL);
     rospf_spoof_net_lsa(intf);
    }
#endif
    nbr->state = NEXSTART;

    nbr->seq = time_sec;
    nbr->I_M_MS = (bit_I | bit_M | bit_MS);
    send_dbsum(intf, nbr, 0);

    msg_event_nbr(nbr, SEQ_MISMATCH, oldstate);
    nbr->events++;

}

static void
NLoadDone __PF2(intf, struct INTF *,
                nbr, struct NBR *)
```

```
{
   u_int oldstate = nbr->state;
   struct AREA *area = intf->area;

   nbr->state = NFULL;
   intf->nbrFcnt++;
   area->nbrFcnt++;
   ospf.nbrFcnt++;

   area->build_rtr = TRUE;
   if (intf->state == IDr)
        BIT_SET(intf->flags, OSPF_INTFF_BUILDNET);

   msg_event_nbr(nbr, LOAD_DONE, oldstate);
   nbr->events++;
}

static void
N1Way __PF2(intf, struct INTF *,
            nbr, struct NBR *)
{
   u_int oldstate = nbr->state;
   struct AREA *area = intf->area;

   rem_hold_tmr(nbr);

   if (nbr->state == NFULL) {
        intf->nbrFcnt--;
        area->nbrFcnt--;
        ospf.nbrFcnt--;

        if ((intf->type <= NONBROADCAST) && (intf->state >= IDr))
          BIT_SET(intf->flags, OSPF_INTFF_NBR_CHANGE);
        else if (intf->type == POINT_TO_POINT || intf->type == VIRTUAL_LINK)
          area->build_rtr = TRUE;

   }
   if (nbr->state >= NEXCHANGE) {
        intf->nbrEcnt--;
        area->nbrEcnt--;
        ospf.nbrEcnt--;
   }
   nbr->state = NINIT;
   nbr->mode = 0;
   nbr->seq = 0;
   nbr->dr = 0;
```

```
    nbr->bdr = 0;


    freeDbSum(nbr);
    REM_NBR_RETRANS(nbr);
    freeLsReq(nbr);
    if (intf->nbrIcnt == 1) {
        freeAckList(intf);
    }

    msg_event_nbr(nbr, ONEWAY, oldstate);
    nbr->events++;

}

static void
NRstAd __PF2(intf, struct INTF *,
             nbr, struct NBR *)
{
    u_int oldstate = nbr->state;
    struct AREA *area = intf->area;

    rem_hold_tmr(nbr);

    if (nbr->state == NFULL) {
        intf->nbrFcnt--;
        area->nbrFcnt--;
        ospf.nbrFcnt--;
    }
    if (nbr->state >= NEXCHANGE) {
        intf->nbrEcnt--;
        area->nbrEcnt--;
        ospf.nbrEcnt--;
    }
    nbr->state = N2WAY;
    nbr->mode = 0;
    nbr->seq = 0;

    freeDbSum(nbr);
    REM_NBR_RETRANS(nbr);
    freeLsReq(nbr);

    if (intf->nbrIcnt == 1) {
        freeAckList(intf);
    }
```

```
   msg_event_nbr(nbr, RST_ADJ, oldstate);
   nbr->events++;


#ifdef ROSPF
   if (intf->type == RADIO_MULTIPOINT)
    (*(nbr_trans[ADJ_OK][nbr->state])) (intf, nbr);
#endif


   if ((intf->type > NONBROADCAST && intf->type <= VIRTUAL_LINK) ||
        ((nbr->state == N2WAY && intf->type < POINT_TO_POINT) &&
         (intf->dr == nbr || intf->bdr == nbr ||
          intf->dr == &intf->nbr || intf->bdr == &intf->nbr)))
        (*(nbr_trans[ADJ_OK][nbr->state])) (intf, nbr);
}


static void
NDown __PF2(intf, struct INTF *,
            nbr, struct NBR *)
{
   u_int oldstate = nbr->state;
   struct NBR *n;
   struct AREA *area = intf->area;

   nbr->state = NDOWN;

#ifdef ROSPF
   if (intf->type == RADIO_MULTIPOINT) {
    if (TRACE_TF(ospf.trace_options, TR_STATE)) {
        trace_only_tf(ospf.trace_options,
                      TRC_NL_AFTER,
                      ("OSPF TRANSITION    Neighbor %A  EVENT %s  %10s -> %-10s",
                       nbr->nbr_addr,
                       "Lower Layer Down",
                       ospf_nbr_states[oldstate],
                       ospf_nbr_states[nbr->state]));
    }
   }
   else {
    msg_event_nbr(nbr, INACT_TIMER, oldstate);
   }
#else
   msg_event_nbr(nbr, INACT_TIMER, oldstate);
#endif
```

```
    nbr->events++;


    rem_hold_tmr(nbr);
    rem_inact_tmr(nbr);


    freeDbSum(nbr);
    REM_NBR_RETRANS(nbr);
    freeLsReq(nbr);
    if (intf->nbrIcnt == 1) {
        freeAckList(intf);
    }

    if (oldstate > NATTEMPT) {
        intf->nbrIcnt--;
        ospf.nbrIcnt--;
        area->nbrIcnt--;
    }
    if (oldstate == NFULL) {
        intf->nbrFcnt--;
        area->nbrFcnt--;
        ospf.nbrFcnt--;
    }
    if (oldstate >= NEXCHANGE) {
        intf->nbrEcnt--;
        area->nbrEcnt--;
        ospf.nbrEcnt--;
    }

#ifdef ROSPF
    if ( intf->type != RADIO_MULTIPOINT)
     ospf_nh_free(&nbr->nbr_nh);
#else
    ospf_nh_free(&nbr->nbr_nh);
#endif


    if (intf->type > BROADCAST) {
        nbr->mode = 0;
        nbr->seq = 0;



#ifdef ROSPF
```

```
        if ( intf->type != RADIO_MULTIPOINT)
          {
#endif

            if (intf->type != VIRTUAL_LINK) {
              if (nbr->nbr_id) {
                  sockfree(nbr->nbr_id);
                  nbr->nbr_id = (sockaddr_un *) 0;
              }
            } else {
              if (nbr->nbr_addr) {
                  sockfree(nbr->nbr_addr);
                  nbr->nbr_addr = (sockaddr_un *) 0;
              }
            }
            nbr->dr = nbr->bdr = 0;
            if (nbr == intf->dr) {
              intf->dr = NBRNULL;
              intf->nbr.dr = 0;
            }
            if (nbr == intf->bdr) {
              intf->bdr = NBRNULL;
              intf->nbr.bdr = 0;
            }

#ifdef ROSPF
          }
#endif
      }
    else {

      if (nbr == intf->dr) {
            intf->dr = NBRNULL;
            intf->nbr.dr = 0;
          }
      if (nbr == intf->bdr) {
            intf->bdr = NBRNULL;
            intf->nbr.bdr = 0;
        }
        for (n = &intf->nbr; n != NBRNULL; n = n->next) {
          if (n->next == nbr) {
                n->next = nbr->next;
                ospf_nbr_delete(intf, nbr);
                nbr = NBRNULL;
#ifdef  notdef
```

```
                ospf.nbr_sb_not_valid = TRUE;
#endif
                break;
            }
        }
        assert(nbr == NBRNULL);
    }

    if ((intf->type <= NONBROADCAST) && (intf->state >= IDr))
        BIT_SET(intf->flags, OSPF_INTFF_NBR_CHANGE);
    else if (intf->type == POINT_TO_POINT || intf->type == VIRTUAL_LINK)
        area->build_rtr = TRUE;
}



#ifdef ROSPF


static void
NRAdjOk __PF2(intf, struct INTF *,
              nbr, struct NBR *)
{
 intf->nbrIcnt++;
 ospf.nbrIcnt++;
 intf->area->nbrIcnt++;

 if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
  trace_tf(ospf.trace_options,
           TR_ROSPF,
           0,
           ("ROSPF: NRAdjOk: transitioning to ADJ_OK for neighbor %A (id %A)",
            nbr->nbr_addr,
            nbr->nbr_id));
 }
 if (!nbr->nbr_nh)
  OSPF_NH_ALLOC(nbr->nbr_nh = ospf_nh_add(nbr->intf->ifap,
                                          NBR_ADDR(nbr),
                                          NH_NBR));

 (*(nbr_trans[ADJ_OK][nbr->state])) (intf, nbr);

}
```

```
_PROTOTYPE(nbr_trans[NNBR_EVENTS][NNBR_STATES],
        void,
        (struct INTF *,
         struct NBR *)) = {

{       NHello,         NHello,         NErr,  NErr,  NErr,  NErr,  NErr,  NErr },
{       NStart, NErr,   NErr,  NErr,  NErr,  NErr,  NErr,  NErr },
{       NErr,  NErr,  N2Way,          NErr,  NErr,  NErr,  NErr,  NErr },
{       NErr,  NErr,  NErr,  NAdjOk,        NErr,  NErr,  NErr,  NErr },
{       NErr,  NErr,  NErr,  NErr,  NNegDone, NErr,       NErr,  NErr },
{       NErr,  NErr,  NErr,  NErr,  NErr,  NExchDone, NErr, NErr },
{       NErr,  NErr,  NErr,  NBadSq,        NBadSq,         NBadSq,         NBadSq,
        NBadSq },
{       NErr,  NErr,  NErr,  NErr,  NErr,  NBadReq, NBadReq, NBadReq },
{       NErr,  NErr,  NErr,  NErr,  NErr,  NErr,  NLoadDone, NErr },
{       NErr,  NErr,  NErr,  N1Way,         N1Way,          N1Way,          N1Way,
        N1Way },
{       NErr,  NErr,  NErr,  NErr,  NRstAd,         NRstAd,         NRstAd,         NRstAd },
{       NDown,          NDown,          NDown,          NDown,          NDown,          NDown,
        NDown,          NDown },
{       NDown,          NDown,          NDown,          NDown,          NDown,          NDown,
        NDown,          NDown },
{       NDown,          NDown,          NDown,          NDown,          NDown,          NDown,
        NDown,          NDown },
{       NRAdjOk,NErr,       NErr,  NErr,  NErr,  NErr,  NErr,  NErr },
};

#else

_PROTOTYPE(nbr_trans[NNBR_EVENTS][NNBR_STATES],
        void,
        (struct INTF *,
         struct NBR *)) = {

{       NHello,         NHello,         NErr,  NErr,  NErr,  NErr,  NErr,  NErr },
{       NStart, NErr,   NErr,  NErr,  NErr,  NErr,  NErr,  NErr },
{       NErr,  NErr,  N2Way,          NErr,  NErr,  NErr,  NErr,  NErr },
{       NErr,  NErr,  NErr,  NAdjOk,        NErr,  NErr,  NErr,  NErr },
{       NErr,  NErr,  NErr,  NErr,  NNegDone, NErr,       NErr,  NErr },
{       NErr,  NErr,  NErr,  NErr,  NErr,  NExchDone, NErr, NErr },
{       NErr,  NErr,  NErr,  NBadSq,        NBadSq,         NBadSq,         NBadSq,
        NBadSq },
{       NErr,  NErr,  NErr,  NErr,  NErr,  NBadReq, NBadReq, NBadReq },
{       NErr,  NErr,  NErr,  NErr,  NErr,  NErr,  NLoadDone, NErr },
```

```
{       NErr,  NErr,  NErr,  N1Way,      N1Way,      N1Way,      N1Way,
        N1Way },
{       NErr,  NErr,  NErr,  NErr,  NRstAd,     NRstAd,     NRstAd,     NRstAd },
{       NDown,      NDown,      NDown,      NDown,      NDown,      NDown,
        NDown,      NDown },
{       NDown,      NDown,      NDown,      NDown,      NDown,      NDown,
        NDown,      NDown },
{       NDown,      NDown,      NDown,      NDown,      NDown,      NDown,
        NDown,      NDown },
};

#endif



/*****************************************************************************
*
 *
 *                    INTERFACE STATE TRANSITIONS
 *

*****************************************************************************
/

static const char *ospf_intf_types[] = {
   "",
   "Broadcast",
   "Nonbroadcast",
   "Point To Point",
   "Virtual"
};

static const char *ospf_intf_events[] = {
   "Interface Up",
   "Wait Timer",
   "Backup Seen",
   "Neighbor Change",
   "Loop Indication",
   "Unloop Indication",

#ifdef ROSPF
   "Interface Down",
   "ROSPF Build Timer"
#else
```

```
   "Interface Down"
#endif
};


const char *ospf_intf_states[] = {
   "Down",
   "Loopback",
   "Waiting",
   "P To P",
   "DR",
   "BackupDR",
#ifdef ROSPF
   "DR Other",
   "Radio Multi-point"
#else
   "DR Other"
#endif
};


#definemsg_event_intf(intf, event, cur) \
        if (TRACE_TF(ospf.trace_options, TR_STATE)) { \
           trace_only_tf(ospf.trace_options, \
                         TRC_NL_AFTER,\
                         ("OSPF TRANSITION %s Interface %A  EVENT %s  %-8s -> %8s", \
                          ospf_intf_types[(intf)->type], \
                          (intf)->type == VIRTUAL_LINK ? (intf)->nbr.nbr_addr: (intf)->ifap-
>ifa_addr, \
                          ospf_intf_events[event], \
                          ospf_intf_states[cur],\
                          ospf_intf_states[(intf)->state]); )\
        }

static void
IErr __PF1(intf, struct INTF *)
{
}

static void
IUp __PF1(intf, struct INTF *)
{

   u_int oldstate = intf->state;
   struct AREA *a = intf->area;
   struct NBR *n;
```

```
    switch (intf->type) {
    case BROADCAST:
        intf->state = IWAITING;
        send_hello(intf, 0, FALSE);
        start_wait_tmr(intf);
        break;

    case POINT_TO_POINT:
        reset_inact_tmr(&intf->nbr);
        intf->state = IPOINT_TO_POINT;
        send_hello(intf, 0, FALSE);
        break;

    case VIRTUAL_LINK:
        intf->state = IPOINT_TO_POINT;
        BIT_SET(intf->trans_area->area_flags, OSPF_AREAF_VIRTUAL_UP);
        ospf.vUPcnt++;
        send_hello(intf, 0, FALSE);
        break;

    case NONBROADCAST:
        if (intf->pri) {
          intf->state = IWAITING;
          start_wait_tmr(intf);
        } else
          intf->state = IDrOTHER;
        for (n = intf->nbr.next; n != NBRNULL; n = n->next)
          (*(nbr_trans[START][n->state])) (intf, n);
        break;


#ifdef ROSPF
    case RADIO_MULTIPOINT:
        {
         if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
          trace_tf(ospf.trace_options,
                   TR_ROSPF,
                   0,
                   ("ROSPF: IUp: intf %A. Adding ourselves to nbr list",
                    intf->ifap->ifa_addr));
         }

         intf->state = IRADIO_MULTIPOINT;

         n = (struct NBR *) task_block_alloc(ospf_nbr_index);
```

```
        n->nbr_id    = sockdup(ospf.router_id);
        n->nbr_addr  = sockdup(intf->ifap->ifa_addr);
        n->intf      = intf;
        ospf_nbr_add(intf,n);

        intf->dr             = n;
        n->dr                = n->nbr_addr;
                trace_tf(ospf.trace_options,
                 TR_ROSPF,
                 0,
                 ("ROSPF: DR initially set to addr %A (ID %A)",
                  intf->dr->nbr_addr,
                  intf->dr->nbr_id));

        if (intf->rospf_spoof_timer) {
          set_rospf_spoof_tmr(intf, intf->rospf_spoof_timer);
        }
        else {
          set_rospf_spoof_tmr(intf, ROSPF_SPOOF_INTERVAL);
        }

        break;
      }
#endif


  }


#ifdef ROSPF
    if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
    trace_tf(ospf.trace_options,
             TR_ROSPF,
             0,
             ("IUp: setting rtr sched, lock time %d cur time %d",
                 ospf.backbone.lock_time, time_sec));
  }
#endif
  if (intf->type != VIRTUAL_LINK)
       a->build_rtr = TRUE;
  else
       set_rtr_sched(&ospf.backbone);

  intf->events++;
```

```
    a->ifUcnt++;
    msg_event_intf(intf, INTF_UP, oldstate);

    intf->up_time = time_sec;
}


static void
IDown __PF1(intf, struct INTF *)
{
    u_int oldstate = intf->state;
    struct NBR *n, *next_nbr;
    struct AREA *a = intf->area;

    intf->state = IDOWN;

    rem_wait_tmr(intf);
    reset_net_sched(intf);
    intf->nbr.dr = 0;
    intf->nbr.bdr = 0;
    intf->dr = NBRNULL;
    intf->bdr = NBRNULL;

    freeAckList(intf);

    msg_event_intf(intf, INTF_DOWN, oldstate);

    for (n = FirstNbr(intf); n != NBRNULL; n = next_nbr) {
        next_nbr = n->next;

#ifdef ROSPF
        if (TRACE_TF(ospf.trace_options, TR_STATE)) {
          trace_only_tf(ospf.trace_options,
                        TRC_NL_AFTER,
                        ("OSPF TRANSITION        Neighbor %A  EVENT %s in state %10s",
                         n->nbr_addr,
                         "Kill Nbr",
                         ospf_nbr_states[n->state]));
        }
#endif
        (*(nbr_trans[KILL_NBR][n->state])) (intf, n);
    }


    if (intf->type <= NONBROADCAST && intf->nbrIcnt == 0) {
        (*(if_trans[NBR_CHANGE][intf->state])) (intf);
```

```
        a->build_rtr = TRUE;
    }
    if (intf->type == VIRTUAL_LINK) {
        BIT_SET(intf->trans_area->area_flags, OSPF_AREAF_VIRTUAL_UP);
        ospf.vUPcnt--;
    }
    a->ifUcnt--;
    intf->events++;
}


static void
ILoop __PF1(intf, struct INTF *)
{
}


static void
IUnLoop __PF1(intf, struct INTF *)
{
}

static void
IWaitTmr __PF1(intf, struct INTF *)
{
    u_int oldstate = intf->state;
    struct AREA *a = intf->area;

    ospf_choose_dr(intf);

    a->build_rtr = TRUE;

    msg_event_intf(intf, WAIT_TIMER, oldstate);

    if (oldstate != intf->state)
        intf->events++;
}



static void
IBackUp __PF1(intf, struct INTF *)
{
    u_int oldstate = intf->state;
    struct AREA *a = intf->area;
```

```
    rem_wait_tmr(intf);

    ospf_choose_dr(intf);

    a->build_rtr = TRUE;

    msg_event_intf(intf, BACKUP_SEEN, oldstate);

    if (oldstate != intf->state)
        intf->events++;
}


static void
INbrCh __PF1(intf, struct INTF *)
{
    u_int oldstate = intf->state;
    struct AREA *a = intf->area;

    BIT_RESET(intf->flags, OSPF_INTFF_NBR_CHANGE);

    if (intf->type > NONBROADCAST)
        return;
    ospf_choose_dr(intf);

    a->build_rtr = TRUE;

    msg_event_intf(intf, NBR_CHANGE, oldstate);

    if (oldstate != intf->state)
        intf->events++;
}




#ifdef ROSPF

void IRNbrCh __PF1(intf, struct INTF *)
{
  boolean_t         isOk;
```

```
 boolean_t        isOldHead;
 nfwd_flag_desc_t * theNode;
 u_int32          theIp;
 u_int32          theMac;
 struct NBR       * theNbr;
 struct LSDB      * theRLA;

#ifdef ROSPF_DEBUG
 if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
  trace_tf(ospf.trace_options,
           TR_ROSPF,
           0,
           ("ROSPF: Retrieving NTDR Forwarding Table"));
 }
#endif


 isOldHead = B_FALSE;
 if ( OSPF_INTFF_RADIO_CLUSTER_HEAD & intf->flags)
  {
   isOldHead = B_TRUE;
  }
 intf->flags = intf->flags & ~OSPF_INTFF_RADIO_CLUSTER_HEAD;



 /*************************************************
        Bring up new adjacencies
 *************************************************/

 theNbr = (struct NBR *) NULL;
 rospf_get_intranet_topology(intf);
 isOk = rospf_get_next_entry(intf,&theNode);
 while (isOk == B_TRUE)
  {
   if (TRACE_TF(ospf.trace_options, TR_ROSPF) &&
        theNode->flags != nfwd_rlflags_CURNODE_MEMBER) {
       trace_tf(ospf.trace_options,
                TR_ROSPF,
                0,
                ("ROSPF: Handling Neighbor %d, flags %x",
                 theNode->destination,
                 theNode->flags));
   }

   if (theNode->flags & nfwd_rlflags_NBR_DIRECT)
```

```
        {
         theIp = ROSPF_CONVERT_MAC_TO_IP(intf,theNode->destination);

         if ( (theNbr = rospf_find_nbr_by_addr(intf,theIp)) == NULL)
          {
           if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
               trace_tf(ospf.trace_options,
                        TR_ROSPF,
                        0,
                        ("ROSPF: Direct Neighbor up %d",
                         theNode->destination));
           }
           theNbr          = (struct NBR *) task_block_alloc(ospf_nbr_index);
           theNbr->nbr_addr   = sockdup(sockbuild_in(0, theIp));
           theNbr->intf       = intf;
           ospf_nbr_add(intf,theNbr);
           theNbr->state      = N2WAY;
           (*(nbr_trans[ADJ_OK][theNbr->state])) (intf, theNbr);
           rospf_save_cluster_role(theNbr,theNode);
           if (theNode->flags & nfwd_rlflags_CURNODE_HEAD)
               {
                intf->flags = intf->flags | OSPF_INTFF_RADIO_CLUSTER_HEAD;
#ifdef ROSPF_DEBUG
                if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
                 trace_tf(ospf.trace_options,
                          TR_ROSPF,
                          0,
                          ("ROSPF: Setting OSPF_INTFF_RADIO_CLUSTER_HEAD for
interface"));
                }
#endif
               }
          }


        else if ( (theNbr->nbr_id !=0 ) &&
                  !(theNbr->nbr_flags & NBR_RADIO_ACTIVE) &&
                  (theNbr->state == NDOWN) )
         {

          if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
              trace_tf(ospf.trace_options,
                       TR_ROSPF,
                       0,
                       ("ROSPF: Direct Neighbor %d up",
                        theNode->destination));
```

```
        }
        if (((theRLA = FindLSA(intf->area,
                               NBR_ID(theNbr),
                               NBR_ID(theNbr),
                               LS_RTR, 0)) != LSDBNULL) &&
            (ADV_AGE(theRLA) < MaxAge))
           {
            if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
             trace_tf(ospf.trace_options,
                      TR_ROSPF,
                      0,
                      ("ROSPF: found Router Links Adv"));
            }
            BIT_SET(theNbr->nbr_flags, NBR_RADIO_ACTIVE);
            intf->rospf_nbr_active_count++;
            BIT_SET(intf->flags, OSPF_INTFF_NEW_NBR_ID_LEARNED);
            trace_log_tf(ospf.trace_options,
                         0,
                         LOG_INFO,
                         ("ROSPF: Nbr %A (id %A) now active",
                            theNbr->nbr_addr,
                            theNbr->nbr_id));
            if (NBR_ADDR(theNbr) < NBR_ADDR(intf->dr)) {
             theNbr->dr = NBR_ADDR(theNbr);
                           trace_log_tf(ospf.trace_options,
                            0,
                            LOG_INFO,
                           ("ROSPF: DR change from nbr addr %A (ID %A) to addr %A (ID
%A)",
                               intf->dr->nbr_addr,
                               intf->dr->nbr_id,
                               theNbr->nbr_addr,
                               theNbr->nbr_id));
             intf->dr = theNbr;
            } else
             theNbr->dr = NBR_ADDR(intf->dr);
            set_rospf_buildnet_tmr(intf, ROSPF_BUILDNET_INTERVAL);
            rospf_spoof_net_lsa(intf);
           }
        if (!theNbr->nbr_nh)
            OSPF_NH_ALLOC(theNbr->nbr_nh = ospf_nh_add(theNbr->intf->ifap,
                                                       NBR_ADDR(theNbr),
                                                       NH_NBR));
        theNbr->state      = N2WAY;
        (*(nbr_trans[ADJ_OK][theNbr->state])) (intf, theNbr);
        rospf_save_cluster_role(theNbr,theNode);
```

```
            if (theNode->flags & nfwd_rlflags_CURNODE_HEAD)
                {
                 intf->flags = intf->flags | OSPF_INTFF_RADIO_CLUSTER_HEAD;
#ifdef ROSPF_DEBUG
                 if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
                  trace_tf(ospf.trace_options,
                           TR_ROSPF,
                           0,
                           ("ROSPF: Setting OSPF_INTFF_RADIO_CLUSTER_HEAD for
interface"));
                 }
#endif
                }
            }


          else if ( (theNbr->nbr_id !=0 ) &&
                    (theNbr->state == NDOWN) )
           {

            if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
                trace_tf(ospf.trace_options,
                         TR_ROSPF,
                         0,
                         ("ROSPF: Indirect Neighbor %d became direct",
                          theNode->destination));
            }
            if (!theNbr->nbr_nh)
                OSPF_NH_ALLOC(theNbr->nbr_nh = ospf_nh_add(theNbr->intf->ifap,
                                                           NBR_ADDR(theNbr),
                                                           NH_NBR));
            theNbr->state      = N2WAY;
            (*(nbr_trans[ADJ_OK][theNbr->state])) (intf, theNbr);
            rospf_save_cluster_role(theNbr,theNode);
            if (theNode->flags & nfwd_rlflags_CURNODE_HEAD)
                {
                 intf->flags = intf->flags | OSPF_INTFF_RADIO_CLUSTER_HEAD;
#ifdef ROSPF_DEBUG
                 if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
                  trace_tf(ospf.trace_options,
                           TR_ROSPF,
                           0,
                           ("ROSPF: Setting OSPF_INTFF_RADIO_CLUSTER_HEAD for
interface"));
                 }
#endif
```

```
                }
            }

            else  if (theNode->flags & nfwd_rlflags_CURNODE_HEAD)
             {
              intf->flags = intf->flags | OSPF_INTFF_RADIO_CLUSTER_HEAD;
#ifdef ROSPF_DEBUG
              if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
                  trace_tf(ospf.trace_options,
                           TR_ROSPF,
                           0,
                           ("ROSPF: Setting OSPF_INTFF_RADIO_CLUSTER_HEAD for
interface"));
              }
#endif
            }

          }
     theNbr = (struct NBR *) NULL;
     isOk   = rospf_get_next_entry(intf,&theNode);
    }


 if ( intf->flags & OSPF_INTFF_RADIO_CLUSTER_HEAD )
   {
    rospf_reset_intranet_topology();
    theNbr = (struct NBR *) NULL;
    isOk = rospf_get_next_entry(intf,&theNode);
    while (isOk == B_TRUE)
        {
         theIp = ROSPF_CONVERT_MAC_TO_IP(intf,theNode->destination);
                 if ( (theNbr = rospf_find_nbr_by_addr(intf,theIp)) == NULL)
           {
            if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
                trace_tf(ospf.trace_options,
                         TR_ROSPF,
                         0,
                         ("ROSPF: Indirect Neighbor up %d",
                          theNode->destination));
            }
            theNbr          = (struct NBR *) task_block_alloc(ospf_nbr_index);
            theNbr->nbr_addr   = sockdup(sockbuild_in(0, theIp));
            ospf_nbr_add(intf,theNbr);
            rospf_save_cluster_role(theNbr,theNode);
           }
```

```
else if ( (theNbr->nbr_id !=0 ) && !(theNbr->nbr_flags & NBR_RADIO_ACTIVE) )
 {
  if (((theRLA = FindLSA(intf->area,
                         NBR_ID(theNbr),
                         NBR_ID(theNbr),
                         LS_RTR, 0)) != LSDBNULL) &&
      (ADV_AGE(theRLA) < MaxAge))
     {
      if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
       trace_tf(ospf.trace_options,
                TR_ROSPF,
                0,
                ("ROSPF: Indirect Neighbor up %d",
                 theNode->destination));
      }

      BIT_SET(theNbr->nbr_flags, NBR_RADIO_ACTIVE);
      intf->rospf_nbr_active_count++;
      BIT_SET(intf->flags, OSPF_INTFF_NEW_NBR_ID_LEARNED);
      set_rospf_buildnet_tmr(intf, ROSPF_BUILDNET_INTERVAL);
      trace_log_tf(ospf.trace_options,
                   0,
                   LOG_WARNING,
                   ("ROSPF: Nbr %A (id %A) now active",
                      theNbr->nbr_addr,
                      theNbr->nbr_id));
      if (NBR_ADDR(theNbr) < NBR_ADDR(intf->dr)) {
       theNbr->dr = NBR_ADDR(theNbr);

       trace_log_tf(ospf.trace_options,
                    0,
                    LOG_WARNING,
                   ("ROSPF: DR change from nbr addr %A (ID %A) to addr %A (ID
%A)",
                      intf->dr->nbr_addr,
                      intf->dr->nbr_id,
                      theNbr->nbr_addr,
                      theNbr->nbr_id));
       intf->dr = theNbr;
      } else
       theNbr->dr = NBR_ADDR(intf->dr);
      set_rospf_buildnet_tmr(intf, ROSPF_BUILDNET_INTERVAL);
      rospf_spoof_net_lsa(intf);
     }
  rospf_save_cluster_role(theNbr,theNode);
 }
```

```
            theNbr = (struct NBR *) NULL;
            isOk   = rospf_get_next_entry(intf,&theNode);
          }
    }


  /************************************************
    tear down old adjacencies or reform changed ones
   ************************************************/
  theNbr  = FirstNbr(intf);
  theNode = NULL;
  while (theNbr != NULL)
    {

     if ( NBR_ADDR(theNbr) == INTF_ADDR(intf))
          {
           ;
          }
     else if ( BIT_TEST(theNbr->nbr_flags,NBR_RADIO_ACTIVE) &&
               ( ((theNode=rospf_find_entry_by_addr(intf,NBR_ADDR(theNbr))) == NULL) ||
                  ( (theNbr->state >= NEXSTART) && !(theNode->flags &
nfwd_rlflags_NBR_DIRECT)) ) )
          {
           trace_log_tf(ospf.trace_options,
                        0,
                        LOG_INFO,
                        ("ROSPF: Nbr %A (id %A) now inactive",
                           theNbr->nbr_addr,
                           theNbr->nbr_id));
           if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
            trace_tf(ospf.trace_options,
                     TR_ROSPF,
                     0,
                     ("ROSPF: Taking Down Adjacency to %d",
                      ROSPF_CONVERT_IP_TO_MAC(intf,NBR_ADDR(theNbr))));
           }

           (*(nbr_trans[LLDOWN][theNbr->state])) (intf, theNbr);
           intf->rospf_nbr_active_count--;
           BIT_RESET(theNbr->nbr_flags,NBR_RADIO_ACTIVE);
           BIT_SET(intf->flags, OSPF_INTFF_NEW_NBR_ID_LEARNED);
           set_rospf_buildnet_tmr(intf, ROSPF_BUILDNET_INTERVAL);
          }
     else if (rospf_is_role_different(theNbr,theNode) == B_TRUE)
          {
           if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
```

```
        trace_tf(ospf.trace_options,
                 TR_ROSPF,
                 0,
                 ("ROSPF: Inconsistency with views of nbr's cluster role, resetting view"));
      }
      rospf_save_cluster_role(theNbr,theNode);
      if (theNode->flags & nfwd_rlflags_NBR_DIRECT )
       {
        if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
            trace_tf(ospf.trace_options,
                     TR_ROSPF,
                     0,
                     ("ROSPF: Resetting Adjacency to %d",
                      ROSPF_CONVERT_IP_TO_MAC(intf,NBR_ADDR(theNbr))));
        }
        (*(nbr_trans[RST_ADJ][theNbr->state])) (intf, theNbr);
       }
     }
 theNbr = theNbr->next;
}


if (!(BIT_TEST(intf->dr->nbr_flags, NBR_RADIO_ACTIVE)) &&
     NBR_ADDR(intf->dr) != INTF_ADDR(intf)) {


 if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
     trace_tf(ospf.trace_options,
              TR_ROSPF,
              0,
              ("ROSPF: searching for new dr"));
 }

 theNbr  = FirstNbr(intf);
 while (theNbr != NULL)
     {
      if ( BIT_TEST(theNbr->nbr_flags,NBR_RADIO_ACTIVE) ||
         NBR_ADDR(theNbr) == INTF_ADDR(intf)) {

        assert(NBR_ADDR(theNbr) != (u_int32) 0);
        theNbr->dr = NBR_ADDR(theNbr);
        trace_log_tf(ospf.trace_options,
                     0,
                     LOG_INFO,
                  ("ROSPF: DR change from nbr addr %A (ID %A) to addr %A (ID %A)",
                     intf->dr->nbr_addr,
```

```
                        intf->dr->nbr_id,
                        theNbr->nbr_addr,
                        theNbr->nbr_id));
            intf->dr = theNbr;
            rospf_spoof_net_lsa(intf);
            break;
           }
           theNbr = theNbr->next;
          }
    }

#ifdef ROSPF_DEBUG
 if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
  trace_tf(ospf.trace_options,
           TR_ROSPF,
           0,
           ("ROSPF: Done Parsing ntdr forwarding table"));
 }
#endif
}


void
IRNLA __PF1(intf, struct INTF *)
{

   u_int oldstate = intf->state;
   struct AREA *a = intf->area;
   struct NBR *n, *next_nbr;
   struct ospf_lsdb_list *trans = LLNULL;
   int rla_flags;

   assert(intf->type == RADIO_MULTIPOINT);

   intf->rospf_buildnet_time = (time_t) 0;
   if (!(BIT_TEST(intf->flags, OSPF_INTFF_NEW_NBR_ID_LEARNED))) {
    trace_log_tf(ospf.trace_options,
                 0,
                 LOG_WARNING,
                 ("ROSPF: processing ROSPF build net timer when
OSPF_INTFF_NEW_NBR_ID_LEARNED flag not set for intf %A(%s)",
                  intf->ifap->ifa_addr,
                  intf->ifap->ifa_link->ifl_name));
   }

   for (n = FirstNbr(intf); n != NBRNULL; n = next_nbr) {
```

```
        next_nbr = n->next;
        if (rospf_build_net_lsa(intf, n, &trans) != FLAG_NO_PROBLEM) {
         if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
          trace_tf(ospf.trace_options,
                   TR_ROSPF,
                   0,
                   ("IRNLA: restart build net timer due to rospf_build_net_lsa error, nbr %A (id
%A)",
                    n->nbr_addr,
                    n->nbr_id));
         }
         restart_rospf_buildnet_tmr(intf,
           ROSPF_BUILDNET_RETRY_MULTIPLIER * intf->retrans_timer);
        }
        else
         intf->area->spfsched |= LS_NET;
        if (trans != LLNULL) {
         if (rospf_self_orig_nla_flood(a, intf, n, trans, LS_NET) ==
            FLAG_NO_PROBLEM) {
          ospf_freeq((struct Q **)&trans, ospf_lsdblist_index);
         }
         else {
          ospf_freeq((struct Q **)&trans, ospf_lsdblist_index);

          if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
           trace_tf(ospf.trace_options,
                    TR_ROSPF,
                    0,
                    ("IRNLA: restart build net timer due to rospf_self_orig_nla_flood error, nbr
%A (id %A)",
                       n->nbr_addr,
                       n->nbr_id));
          }
          restart_rospf_buildnet_tmr(intf,
            ROSPF_BUILDNET_RETRY_MULTIPLIER * intf->retrans_timer);
         }
        }
  }



  if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
   trace_tf(ospf.trace_options,
            TR_ROSPF,
            0,
```

```
            ("IRNLA: calling build_rtr_lsa, trans %x",
               &trans));
    }
    rla_flags = build_rtr_lsa(a, &trans, 0);
    intf->area->spfsched |= rla_flags;
    if (trans != LLNULL) {
     self_orig_area_flood(intf->area, trans, LS_RTR);
     ospf_freeq((struct Q **)&trans, ospf_lsdblist_index);
    }

    if (intf->rospf_nbr_active_count > 0) {
     if (BIT_TEST(intf->flags, OSPF_INTFF_ROSPF_TRANS_NET_BUILT) &&
          rla_flags == FLAG_NO_PROBLEM) {
         if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
          trace_tf(ospf.trace_options,
                   TR_ROSPF,
                   0,
                   ("IRNLA: calling rospf_spoof_build_net_lsa, trans %x",
                    &trans));
         }
         if (rospf_spoof_build_net_lsa(intf->area, intf, &trans, TRUE) !=
           FLAG_NO_PROBLEM) {

          if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
           trace_tf(ospf.trace_options, TR_ROSPF, 0,
                    ("ROSPF: IRNLA: defer build_rtr_lsa due to rospf_spoof_build_net_lsa
error"));
          }
          assert(intf->rospf_spoof_time != 0);
         }
         if (trans != LLNULL) {
          intf->area->spfsched |= NETSCHED;
          rospf_area_flood(intf->area, intf, trans);
          ospf_freeq((struct Q **)&trans, ospf_lsdblist_index);
         }
     }
     else if (!(BIT_TEST(intf->flags, OSPF_INTFF_ROSPF_TRANS_NET_BUILT))) {
         if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
          trace_tf(ospf.trace_options, TR_ROSPF, 0,
                   ("rospf_spoof_net_lsa: defer build_rtr_lsa due to not transit net, build_rtr_lsa
return %x",
                        rla_flags));
         }
         assert(intf->rospf_spoof_time != 0);
     }
     else if (rla_flags != FLAG_NO_PROBLEM && rla_flags != RTRSCHED) {
```

```
        if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
          trace_tf(ospf.trace_options,
                   TR_ROSPF,
                   0,
                   ("ROSPF: IRNLA: spoof failed due to build_rtr_lsa return %x",
                    rla_flags));
        }
        assert(intf->rospf_spoof_time != 0);
      }
    }

    if (!intf->rospf_buildnet_time) {
      BIT_RESET(intf->flags, OSPF_INTFF_NEW_NBR_ID_LEARNED);
    } else {
      if (TRACE_TF(ospf.trace_options, TR_ROSPF)) {
        trace_tf(ospf.trace_options,
                 TR_ROSPF,
                 0,
                 ("IRNLA: build net timer restarted value %d",
                  intf->rospf_buildnet_time));
      }
    }

    if (intf->area->spfsched)
      ospf_spf_sched();

    intf->events++;
    msg_event_intf(intf, RBUILD_TM, oldstate);
}

_PROTOTYPE(if_trans[NINTF_EVENTS][NINTF_STATES],
           void,
           (struct INTF *)) = {

{ IUp,  IErr,    IErr,   IErr, IErr,  IErr,   IErr,   IErr },
{ IErr, IErr,    IWaitTmr,IErr, IErr,  IErr,   IErr,   IErr },
{       IErr, IErr,    IBackUp, IErr, IErr,  IErr,   IErr,   IErr },
{       IErr, IErr,      IErr,   IErr, INbrCh,INbrCh, INbrCh,  IRNbrCh },
{       ILoop,IErr,    IErr,    IErr, IErr,  IErr,   IErr,   IErr },
{       IErr, IUnLoop,  IErr,   IErr, IErr,  IErr,   IErr,   IErr },
{       IErr, IDown,   IDown,          IDown,IDown, IDown,  IDown,   IDown },
{ IErr,IErr,    IErr,   IErr, IErr,  IErr,   IErr,   IRNLA },
};

#else
```

```
_PROTOTYPE(if_trans[NINTF_EVENTS][NINTF_STATES],
        void,
        (struct INTF *)) = {
{       IUp,    IErr,   IErr,   IErr,   IErr,   IErr,   IErr },
{       IErr,   IErr,   IWaitTmr, IErr,        IErr,   IErr,   IErr },
{       IErr,   IErr,   IBackUp, IErr,IErr,    IErr,   IErr },
{       IErr,   IErr,   IErr,   IErr,   INbrCh,         INbrCh,         INbrCh },
{       ILoop, IErr,    IErr,   IErr,   IErr,   IErr,   IErr },
{       IErr,   IUnLoop,IErr, IErr,   IErr,   IErr,   IErr },
{       IErr,   IDown,          IDown,          IDown,          IDown,          IDown,          IDown
}
};

#endif
```